DEVELOPING A MINI-APP FOR EXPLORING ALGORITHMS FOR UNSTRUCTURED MESH DETERMINISTIC DISCRETE ORDINATES TRANSPORT ON MANY-CORE ARCHITECTURES

Tom Deakin[1], **Simon McIntosh-Smith**[1], **Justin Lovegrove**[2], **Richard Smedley-Stevenson**[2], **and Andrew Hagues**[2] *
[1]University of Bristol
Bristol, UK
tom.deakin@bristol.ac.uk, S.McIntosh-Smith@bristol.ac.uk

[2]Atomic Weapons Establishment
Aldermaston, UK
{first.last}@awe.co.uk

ABSTRACT

Recent trends in computational architecture design are yielding processors with deep and complex memory hierarchies consisting of small capacity caches and large capacity main memory. CPU parallelism is also hierarchical, consisting of SIMD vector units contained within multiple computational cores with one or more packages in a multi-socket system. Solving the deterministic discrete ordinates transport equation effectively on these architectures requires extracting and effectively mapping concurrent work to the processing elements to leverage performance close to the maximum attainable. This challenge becomes more acute when an unstructured spatial domain is required, where the sweep dependency between neighbouring spatial cells/elements is not implicit as for a structured grid. In this paper we introduce the transport community to the UnSNAP mini-app, a port of the well known SNAP proxy application. UnSNAP was developed to investigate the performance of arbitrarily high-order discontinuous Galerkin finite element unstructured deterministic transport codes on advanced architectures. Approaches to local matrix assembly and solution are evaluated in order to assess their performance for different element orders, and discuss the trade-offs with respect to performance and memory capacity limits of advanced architectures. The performance limiting factors will be explored on many-core architectures, including CPUs from Intel, AMD and Marvell (Arm). We will also discuss performing unstructured sweeps on GPU devices highlighting the associated challenges.

KEYWORDS: Mini-App, S_N transport, sweeps, unstructured

1. INTRODUCTION

The solution of the deterministic discrete ordinates Boltzmann transport equation is computationally expensive. Indeed, it was observed that 50–80% of simulation time on Department of Energy supercomputers is taken by solving transport [1]. The inversion of the streaming-collision operator typically dominates the runtime of the solver. This operation introduces an upwind dependency resulting in the requirement for a sweep across the spatial mesh for each discrete ordinates direction. The high dimensionality of the solution demands a large memory footprint often utilising close to the available memory capacity of the system.

The use of unstructured meshes has gained importance within this community. An unstructured mesh can offer better representation of the problem domains, however the connectivity between neighbouring elements must be explicitly defined. As such, the sweep dependency must be calculated directly as the wavefront is no longer implied simply from structured cell coordinates. Additionally, a different spatial discretisation method must be used as the finite difference method (used for a structured mesh) is no longer directly applicable. For our study, we use a discontinuous Galerkin finite element discretisation of the spatial domain. Lagrange elements are used of arbitrary order, with the 3D spatial basis functions constructed from tensor products of 1D basis functions. The method is considered 'matrix-free' as the large global matrix is not assembled; the local matrices associated to each element are assembled and solved directly. To practically investigate the performance of such a finite element transport code, SNAP has been ported to use this method, with the port nicknamed *UnSNAP* [2]. Those unfamiliar with SNAP and the discretisations, data and iterative schemes are recommended to consult the technical report that accompanies the proxy app [3] or the thesis of the lead author [4].

UnSNAP provides the vehicle for serious study into the performance of unstructured transport on current and future supercomputing architectures. This paper makes the following contributions:

- The performance of assembling the local matrix on-the-fly is compared to precomputing the inverse of the local matrix. We explore this for varying element order and assess the feasibility of this approach on modern many-core architectures with respect to memory footprint.
- The performance limiting factors of unstructured sweeps are investigated on many-core CPU architectures from multiple vendors: Intel, IBM and Marvell (Arm).
- We present porting efforts of unstructured sweeps to GPU architectures and highlight which techniques can give a viable path to exploit this architecture for this algorithm.

2. THE UNSNAP MINI-APP

The UnSNAP mini-app has been developed to explore the performance of solving the transport equation on 3D unstructured meshes using the finite element method [2]. It has been formed as a port of the open-source SNAP proxy application from Los Alamos National Laboratory: a performance proxy for solving transport on structured grids. The stationary transport equation solved by our UnSNAP mini-app is given in Eq. (1), as taken from the SNAP proxy application. The application of the finite element method to Eq. (1) has been omitted for brevity. The total cross section σ and scattering cross section σ_s data and external source q_{ex} are taken directly from SNAP,

where their values are constant at all finite element nodes within a cell. Note that the sources computed from the cross section data and scalar flux *do* vary with spatial location and so in UnSNAP the sources are calculated for each spatial node in the element; extending to spatially differing cross sections within an element is a straightforward extension to the source routines which would have limited impact on our conclusions in this work. The iterative scheme is formed from simple iterations on the scattering source (the right-hand side of Eq. (1)), using Jacobi iterations for the group-to-group coupling in the source as described by Baker [5]. As such, as in SNAP, energy groups can be swept concurrently and this assumption is maintained in UnSNAP. Angles within each octant are swept concurrently in SNAP and this is one source of concurrency in a 3D unstructured mesh too; however we have found that parallelising this dimension on multi-core CPUs results in a prohibitively costly atomic reduction to compute the scalar flux from the angular flux and hence the angular dimension is serialised in UnSNAP [2]. The element nodes form an additional level of concurrency in the assembly and solution of the local linear system that forms the heart of the finite element solve for each element/group/angle. Further details on the assembly are found in Section 4. It is important to note that although we do not include the time-dependent term (and the resulting diamond difference update), the angular flux ψ is still stored in its entirety. Note too that the SNAP mini-app only performs the diamond difference update to the angular flux in time once per timestep (once the source terms have converged) and so we do not include this detail for simplicity.

$$\hat{\Omega}\cdot\vec{\nabla}\psi(\vec{r},\hat{\Omega},E)+\sigma(\vec{r},E)\psi(\vec{r},\hat{\Omega},E) = q_{\text{ex}}(\vec{r},\hat{\Omega},E)+\int dE' \int d\Omega'\sigma_{\text{s}}(\vec{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\psi(\vec{r},\hat{\Omega}',E') \tag{1}$$

An upwind sweep schedule is constructed and followed to organise the compute within each computational node (each MPI rank). OpenMP threads are used for parallelism. Each sweep schedule corresponding to an angular direction is executed in turn (serially) to remove the need for synchronisation in updating the scalar flux in parallel. Threads are used to parallelise across the energy group domain and the elements within each sweep with upwind conditions satisfied — that is those elements along each wavefront. Automatic compiler vectorisation is used (with the help of OpenMP's `simd` directives) along the nodes within each element. For example, when assembling the matrix for the element/group/angle currently being computed, SIMD instructions operate in parallel on a row of the matrix. The Gaussian Elimination step also uses SIMD instructions to update each row of the matrix.

An unstructured mesh data structure is used to maintain the position of element nodes and element connectivity, in particular the cell-to-cell connectivity and the mappings between nodes, faces and cells. A one-dimensional index is assigned to each element as a labelling scheme, and so all data arrays with a spatial component use this index. Each element is a hexahedral element with multiple nodes according to the chosen element order. Curved elements are allowed, although the upwinding along a curved face is averaged which is an approximation deemed appropriate for the scope of this work. In order to strike the appropriate balance for proxy applications, the mini-app does not use any interpolation of fluxes during upwinding which would occur in the case of multiple neighbouring elements such as in an AMR mesh. It would be simple to add this to the mini-app, however this is an unnecessary complication for our work on the performance of the method. The unstructured mesh is populated with the SNAP meshes, with twists applied to ensure

that the mesh is not perfectly structured. Note that it is important to remember that at no point are any of the assumptions about a structured mesh utilised within the algorithms implemented and therefore none of the 'shortcuts' that could be used in a relatively structured mesh are employed; as such our approaches apply to unstructured meshes in generality. We use the spatial decomposition as recommended by Pautz and Bailey to distribute the mesh between MPI processors; an approach which was often shown to provide the best performance for unstructured meshes [6].

2.1. PERFORMANCE RESULTS ON MANY-CORE

In order to provide context to the work described in this paper, this section presents and discusses performance results for the UnSNAP mini-app on a range of many-core architectures. Results are presented on Intel Xeon, Marvell ThunderX2 and IBM Power CPU processors, as well as early results on NVIDIA GPUs. This selection of architectures represents the cutting edge HPC processor technologies available today.

The theoretical peak performance capabilities of these processors are detailed in Table 1. The GW4/EPSRC supercomputer, 'Isambard', was used to collect results on all processors with the exception of Skylake. We used Intel Skylake processors from the Cray XC50 supercomputer, 'Swan'. We used the Cray compiler for the Broadwell and ThunderX2, the Intel compiler for Skylake, and GCC for the Power 9.

Table 1: Processor peak performance information

Architecture	Cores	Clock GHz	Peak FP64 FLOP/s	Main memory bandwidth GB/s
Intel Broadwell	18×2	2.1	1.21	154
Intel Skylake	28×2	2.1	3.76	256
Marvell ThunderX2	32×2	2.5	1.28	288
IBM Power 9	20×2	3.2	1.02	340
NVIDIA P100	60 SMs	1.13	4.04	732
NVIDIA V100	84 SMs	1.37	7.01	900

Figure 1 shows the solve time of UnSNAP running on a variety of architectures for different finite element orders compared to the baseline performance achieved on the Broadwell processors (5.33s, 21.66s and 177.09s for the increasing orders). The problem consisted of 512 cells, 80 angles and 32 energy groups representing 1.3 million DoF (degrees of freedom) in the transport equation, and as we are solving for the angular flux the DoF is multiplied by the number of nodes in the elements; for linear elements there are 10.5 million DoF, quadratic 35.4 million DoF and cubic 83.9 million DoF. A single time step was run, with up to 15 outer and 5 inner iterations per outer under SNAP's source iteration scheme with a convergence of `1E-4`. One OpenMP thread was used per physical core on the CPU platforms.

For all element orders, the Intel Xeon Skylake processors provide the fastest runtime offering between 1.5–4.1X speedup over the Intel Broadwell processors. The Marvell ThunderX2 processor

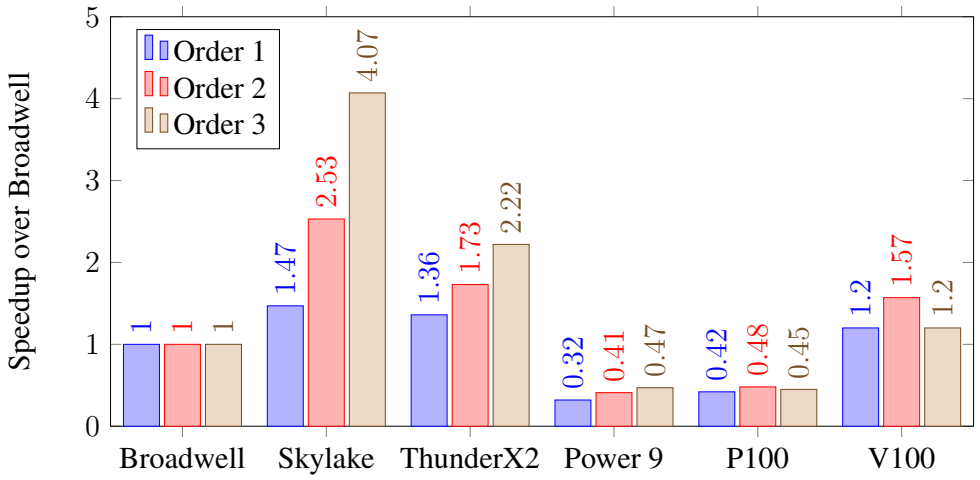

Figure 1: Relative speedup of the solve on a variety of architectures

provides the second best performance for all element orders, and for linear elements the performance is similar to Skylake. At higher orders, ThunderX2 is still 2X faster than Broadwell but does not match the performance of Skylake. The Power 9 processor does not provide a performance improvement over the Broadwell processor for any element orders tested for our input problem. Section 3 will explore performance bounds of the algorithm further as simple conclusions cannot be drawn comparing the achieved speedups to relative performance increases of the processors in Table 1.

We include the results of our early efforts in porting UnSNAP to GPU and it is clear that further optimisation is required. UnSNAP was ported to CUDA so that precise control of the parallelism could be exposed through the 'threads' and 'thread-blocks' available in that programming model. The concurrent scheme for this first version has the same concurrency as is exposed in the CPU version. CUDA threads are used in place of the vectorised loops of the CPU version; that is each thread-block contains threads equal to the number of finite element nodes in each element. The threads concurrently assemble and update rows of the matrix. A kernel is launched for each wavefront level in the sweep schedule, which maintains the strict sweep dependency between the cells. Each kernel is launched with the number of thread-blocks set to the product of the number of energy groups and the number of cells on that level of the sweep schedule graph. The number of thread-blocks is equal to the number of iterations operated on by OpenMP threads in the CPU version. The other kernels in UnSNAP were also ported to CUDA so that the data can remain resident in device memory to reduce host-to-device transfers.

The P100 in particular does not show promising performance, however the V100 does offer some performance speedups although less than for the CPU architectures. The peak performance of the metrics shown in Table 1 indicate that faster runtimes are indeed expected on this architecture if bound by either main memory bandwidth or floating-point operations. The bound is not this simple on CPU architectures, as we will discuss in Section 3.

In profiling the GPU implementation, it is clear that there is insufficient parallelism exposed with

the approach described above. In particular there are too few 'threads' per 'thread-block' to provide enough work within each block, especially for linear elements where only 8 threads are used. On the NVIDIA Pascal architecture, threads will execute in a lock-stepped batch of 32 (known as a 'warp') and so the approach clearly underutilises the device; the Volta architecture offers improvements in this area by integrating program counters per thread thus increasing the ability to progress partial warps. As such, the GPU implementation requires further research and we are currently developing algorithmic changes which may better suit a GPU architecture. The agility of UnSNAP as a mini-app provides an ideal vehicle in which to explore this space.

3. PERFORMANCE ANALYSIS

The results shown in Section 2.1 do not correlate with the commonplace performance bounds. The (cache aware) Roofline model defines computational intensity as the number of floating-point operations per byte of memory read [7]. The nature of the compute kernel primarily consists of assembling a small linear system from a number of different arrays, and then solving the system with Gaussian elimination. The assembly section of this routine has a low operational intensity, collating data from a number of arrays to form rows of the matrix. The Gaussian elimination step performs simple multiply-add operations on each row of the matrix, and contains one division operation per row. The size of the linear system is determined by the finite element order, and so the systems are relatively small for our purposes. As such, the whole routine will not be considered "floating-point bound" under the Roofline model on any modern architecture for the element orders considered in this study. Therefore the Roofline model would be classify the kernel as "memory bandwidth bound".

Modern CPU architectures are complex, hierarchical constructions of computational units and memory caches. It is also common to then design a supercomputer with two sockets per physical node, resulting in NUMA regions. The processor design is usually formed from replication of a 'core', connected to other cores and some shared infrastructure using a network-on-chip. It is common that the core consists of the functional units (for our purposes we consider the floating-point pipeline) and one or two levels of cache. Adding more cores therefore increases the peak floating-point performance of the processor, but perhaps less commonly thought of, more cores increases the capacity and aggregate bandwidth of the closest levels of the memory hierarchy. However, adding more cores to a single processor does not increase the performance of the shared infrastructure, and for this study we consider this to be the last level cache (LLC) and main memory bandwidth. Adding a second socket does clearly increase the memory bandwidth of the node. It is important to note that main memory bandwidth is saturated on these processors when not all the cores of a socket are utilised.

We are fortunate that the high core count of today's processors allows us to perform the following experiment, as suggested by Voysey [8]: (1) Run the code using all cores of a single socket; then (2) Run the code using half the cores of two sockets. Practically this requires first utilising all physical cores of a single socket using the appropriate number of OpenMP threads, and then evenly distributing these threads across the two sockets. Both these runs use the same number of physical cores, and so the total amount of 'core' resources is identical. Specifically, the peak floating-point performance provided is the same, as is the bandwidth and capacity of the close caches. However, in the second case, there are more external resources in the form of LLC capacity and main memory

bandwidth. For a code bound by the main memory bandwidth, one would expect that the second run therefore is significantly faster than the first.

We performed this experiment on the UnSNAP mini-app on the CPU processors using the same input as before. The 'numactl' tool was used to select the cores for each configuration. Our results are shown in Fig. 2, detailing the relative performance increase of the two-socket experiment compared to the one-socket experiment.

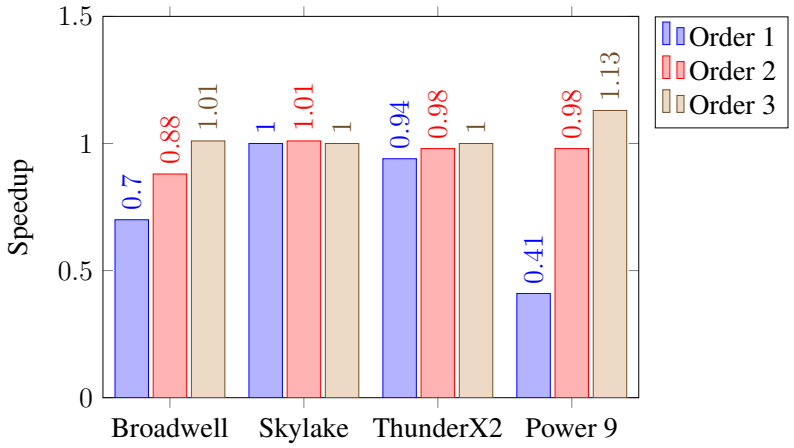

Figure 2: Relative solve time using identical thread counts on one socket fully populated and two sockets each half populated.

For the vast majority of the results parity runtimes are seen. This clearly indicates that the application is bound by on-core resources, with the increased out-of-core resources providing no benefit. As such, the main memory bandwidth is not the performance limiting factor for any of the orders tested on these CPU architectures.

For linear order elements on the Broadwell and Power 9 processors, the experiment shows that adding in an additional socket reduces the performance significantly. Using the CrayPAT performance profiling tool on the Broadwell system, the number of L2 cache hits is severely reduced from 53% to 7% hits. Note that in both runs, the L1 hit rate is exceptionally high at over 95% hits.

One must also expect NUMA effects to come into play, and that can be seen by observing the total memory allocated to each socket is not even. The transport algorithm requires data arrays of different extents and dimensions, not all of which operate over the parallel domain (for example S_N angular quadrature). These arrays will be allocated and initialised in one NUMA domain, however are small enough to exist in the caches of each socket; however if they fall out of cache a latency penalty for reading them from a different NUMA region will apply.

In conclusion, one must turn to the features contained within the core architecture for a performance limiting factor to unstructured transport. The incredibly high L1 data cache hit rate will mean that almost all the memory accesses will be served by these caches. The relative aggregate cache bandwidths for the CPUs architectures tested do not correlate well with the results in Fig. 1 [9], and so L1 cache *bandwidth* alone cannot be attributed to be the limit on performance.

The cache systems and the differences between them for the architectures used in this study are complicated to characterise. Therefore, whilst we have shown that the performance limiting factor for unstructured transport on CPU architectures lies within the cache hierarchy, the exact nature of this bound is a topic for future study.

4. APPROACHES TO LOCAL MATRIX ASSEMBLY

When solving the transport equation (1) using source iterations, a 'global matrix-free' finite element discretisation requires the construction and solution of a small linear system for each angle-element-group. These systems are constructed and solved following the upwind sweep dependency for each angular direction, and utilise nodal angular flux data from upwind neighbouring elements. We explore two approaches: (1) 'on-the-fly' assembly where the system is assembled from its constituent operators, solved and discarded; and (2) where the local matrices are assembled, inverted and stored ahead of the sweep.

The integration of multiplicative pairs of the basis functions $f_i f_j$ (and their derivatives $f_i \partial f_j/\partial x$, $f_i \partial f_j/\partial y$ and $f_i \partial f_j/\partial z$) are always precomputed at the start of each timestep *for each element* using Gaussian quadrature rules over the volume and the faces; essentially these are the mass matrix, volumetric gradients and face matrices. These are computed and stored each timestep. Similarly, the normals to each face are precomputed and stored within the mesh data structure.

The first approach we explore ('on-the-fly' assembly) is to construct the linear system from the streaming-collision operator and upwind angle/normal coefficients ($\hat{\Omega} \cdot \vec{n}$), and the right-hand-side vector from the source term and upwind angular flux data. As such, the 'on-the-fly' assembly requires constructing the system from the already computed small basis pair arrays, problem data (S_N quadrature directions and total cross sections) and the latest source term in the iterative scheme. Additionally neighbouring angular fluxes are required for the upwinding dependency.

This system is then solved directly using Gaussian Elimination to compute the updated angular flux and subsequently the scalar flux required for the next source iteration. Although Gaussian Elimination is well known for limited numerical stability due to floating-point round-off after division by small numbers, this issue has not been noticed within UnSNAP. We explored more stable matrix factorisation approaches for solution using Intel's Math Kernel Library, but they have reduced performance in this context [2]. This may be a result of a lack of optimisation within the library for the small matrices, but also no reuse of the factored matrix within the algorithm; a property often relied on to offset the cost of factorisation in such linear algebra libraries. In the same study, we explored the relative cost of assembling the matrix versus solving the system: for linear elements 34% of the time was in the solve, increasing to 75% for higher orders.

The alternative approach that we explore ('precomputed') in this paper leverages the fact that only the right-hand side of the linear system need be updated for each source iteration. The left-hand side matrix only requires cross section and S_N quadrature data and so is not dependent on the iterative scheme for updating the source. The right-hand vector on the other hand requires updating every source iteration (as the source has changed) and the upwinded angular flux term which is updated as the sweep progresses which contributes to the next source update in the subsequent source iteration. As such, the left-hand side matrix for each angle-element-group may be constructed and inverted only once per timestep, before the source iterations commence, at the expense of increased

memory footprint. This leads to a much simpler computational kernel within the sweep to compute the new fluxes: building the right-hand side vector from the source and upwind neighbours and performing a matrix-vector multiplication with the already inverted matrix.

Results comparing these methods are shown in Fig. 3 for varying finite element order running on the Intel Broadwell system using the Cray compiler 8.6.4. The problem size used is the same as that earlier in this study. The left bars show the first approach of assembling and solving the system on-the-fly, and the right bars showing the second approach of precomputing inverses transforming the sweep kernel to mainly matrix-vector operations. The results show that precomputing the inverse in advance is an attractive solution in terms of runtime. For the main computational work of assembling and inverting a linear system the precomputed approach is 1.9X, 2.1X and 3.2X faster for first, second and third order finite elements respectively. This is a significant improvement for all element orders tested. These improvements are also seen across the CPU architectures tested which can be seen in Fig. 4 which shows the improvements in solve time using the precomputed inverse approach. It is future work to explore this approach on GPUs.

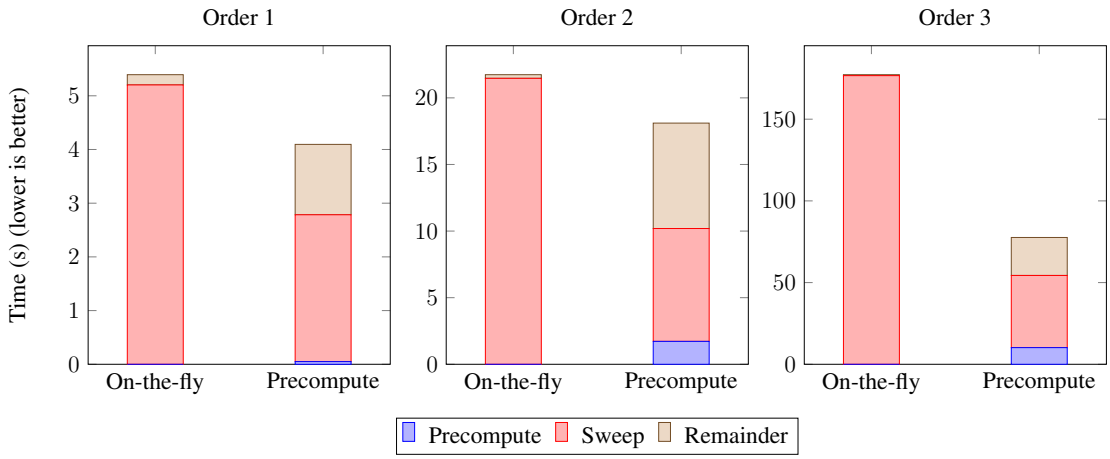

Figure 3: Performance of local assembly methods: on-the-fly (left) and precompute (right)

The improvements observed can be attributed to better data reuse. Constructing the matrices once per timestep as they are invariant across the scattering source iteration scheme ensures that the many data arrays (S_N quadrature, cross sections and basis functions) are read less often thus reducing the memory movement. Computing the inverses unlocks much more regular parallelism than is available in the sweep too, and allows for easier reuse of the shared data: the basis functions of a given element are the same for all energy groups and so this data can be reused. The precomputation step itself is still relatively cheap compared to the sweep operation, taking only up to 25% of the sweep time for the higher-order elements. In a large calculation utilising multiple computational nodes, each node may perform this precompute step in parallel as it requires no communication ("embarrassingly parallel"), and the sweep time will likely increase due to becoming bound by the network [10]. Importantly too, note that we now no longer perform a linear solve during the sweep and only need perform a small matrix-vector multiplication which is much cheaper. Indeed, the inversion which uses a similar Gaussian elimination method to the solve is now only performaned in the precompute steps. Thus we see a larger improvement in runtime than

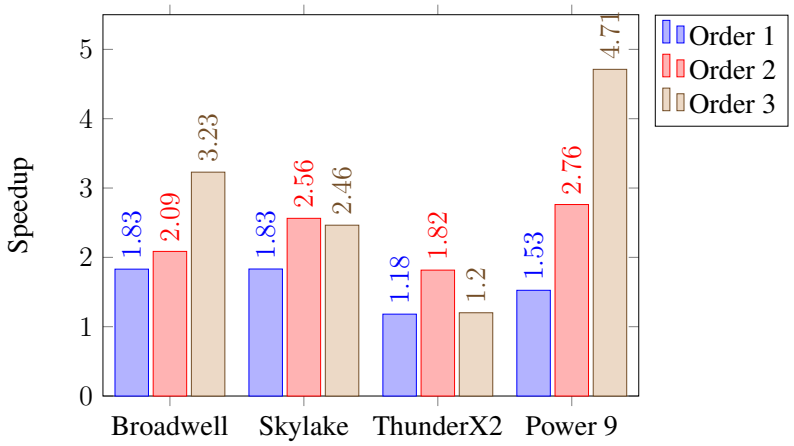

Figure 4: Relative improvement of inverse precomputation vs on-the-fly

just elimination of the cost of the solve during the sweep (recall this was up to 75% as above).

Some extra overhead is present in the current implementation of UnSNAP for the precomputed method, as seen in the 'Remainder' part of Fig. 3. This is from the memory allocation of the inverted matrices where in the current version many small allocations are used, but it is a simple optimisation to reduce this overhead by using a single bulk allocation instead.

These improvements come at a memory footprint cost for the storage of the inverted matrices for each element, group and angle. This creates a memory requirement larger than the angular flux — the number of nodes per element times larger at 8, 27 and 64 times larger for 1st–3rd order. For the problem in this paper, the storage requirements for the precomputed matrices are 0.67 GB, 7.64 GB and 42.95 GB for 1st–3rd order respectively.

This is a significant memory capacity requirement when compared to the capacity offered by memory technology on many-core devices. Devices such as the V100 GPU offer 32 GB of HBM2 (high-bandwidth memory), and we predict that other devices based on similar technology will offer only modest increases in capacity. Therefore, for third order with around 1 million transport DoF per node, the precomputed approach is not viable on such architectures simply due to the memory footprint. Fortunately, it is common in the meshing community to use high order elements in order to reduce the spatial resolution therefore reducing the number of cells and subsequently the number of DoF. Some mesh coarsening will be required in order to fit the capacity constraints of modern many-core architectures. This is difficult to explore within the context of a mini-app as the mesh design is often problem dependent and so we have to defer this to future work. For low order elements, and in particular second-order, the increased memory footprint from storing the inverted matrices is likely to be a palatable trade-off for the performance improvements.

5. CONCLUSIONS

Solving the deterministic transport equation on unstructured meshes on modern many-core architectures demonstrates a number of challenges. We have used the UnSNAP mini-app to study the

performance of such methods on many-core CPUs from a variety of vendors and have begun our study into its performance on GPU architectures. Precomputing and storing the inverse of the finite element matrix offers some reasonable performance improvements on all CPU architectures, most noticeably for second-order elements. However, this comes at the expense of an increase in memory footprint. Optimising for the data movement at the expense of increased footprint is practical at low orders, but for the highest order meshes this may be infeasible given the memory capacity constraints of the memory technologies being introduced into the processors. We have explored the performance limiting factors on CPU architectures. The typical bounds of main memory bandwidth and floating-point operations do not bound the performance of the application. Rather the bound is found in the cache hierarchy for all element orders, which is unusual for an application with such a large memory footprint that exceeds cache sizes.

REFERENCES

[1] A. Hoisie, O. Lubeck, and H. Wasserman. "Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications." *International Journal of High Performance Computing Applications*, **volume 14**(4), pp. 330–346 (2000).

[2] T. Deakin, S. McIntosh-Smith, J. Lovegrove, R. Smedley-Stevenson, and A. Hagues. "UnSNAP : a mini-app for exploring the performance of deterministic discrete ordinates transport on unstructured meshes ." In *Cluster Computing, 2018 IEEE International Conference on*, pp. 566–574. IEEEE, Belfast (2018).

[3] R. J. Zerr and R. S. Baker. "SNAP: SN (Discrete Ordinates) Application Proxy - Proxy Description." Technical report, LA-UR-13-21070, Los Alamos National Labratory (2013).

[4] T. Deakin. *Leveraging Many-Core Technology for Deterministic Neutral Particle Transport at Extreme Scale*. Ph.D. thesis, University of Bristol (2018).

[5] R. S. Baker. "An Sn Algorithm for Modern Architectures." In *Joint International Conference on Mathematics and Computation, Supercomputing in Nuclear Applications and the Monte Carlo Method*, ANS MC2015. American Nuclear Society, Nashville, TN (2015).

[6] S. D. Pautz and T. S. Bailey. "Parallel Deterministic Transport Sweeps of Structured and Unstructured Meshes with Overloaded Mesh Decompositions." *Nuclear Science and Engineering*, **volume 185**(1), pp. 1–17 (2017).

[7] A. Ilic, F. Pratas, and L. Sousa. "Cache-aware Roofline model: Upgrading the loft." *IEEE Computer Architecture Letters*, **volume 13**(1), pp. 21–24 (2014). URL http://ieeexplore.ieee.org/document/6506838/.

[8] A. Voysey and M. Glover. "Performance of Met Office Weather and Climate Codes on Cavium ThunderX2 Processors." (2018). URL https://www.youtube.com/watch?v=xSLY0RJBEAQ. Presentation at Arm Research Summit, Austin, Texas.

[9] M. Martineau, P. Atkinson, and S. Mcintosh-Smith. "Benchmarking the NVIDIA V100 GPU and Tensor Cores." In *HeteroPar workshop at EuroPar International Conference on Parallel and Distributed Computing*. Turin, Italy (2018).

[10] T. Deakin, S. McIntosh-Smith, and W. Gaudin. *Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale*, pp. 429–448. Springer International Publishing, Cham (2016).